Devising a Framework for Efficient and Accurate Matchmaking and Real-time Web Communication

Charles Li
Dartmouth College

Devising a framework for efficient and accurate matchmaking and real-time web communication

Charles Li
Department of Computer Science
Dartmouth College
Hanover, NH
charles.li.16@dartmouth.edu

***Abstract*—Many modern applications have a great need for matchmaking and real-time web communication. This paper first explores and details the specifics of original algorithms for effective matchmaking, and then proceeds to dive into implementations of real-time communication between different clients across the web. Finally, it discusses how to apply the techniques discussed in the paper in practice, and provides samples based on the framework.**

Keywords—matchmaking; real-time; web

I. INTRODUCTION (*MOTIVATIONS*)

A large number of modern applications ("apps") share a common theme: inter-user interaction. There are two large categories of apps in particular that require such multi-user capabilities. The first is **social networking** apps. This comprises both apps designed for online dating, and also apps designed to cultivate non-romantic relationships, such as friendships or professional relationships. Popular examples include Facebook, LinkedIn, and Tinder.

Matchmaking is a key component in such social apps, given their essential need to "connect" individuals with others. What might make an individual compatible with another includes a variety of factors. This might include interests and mutual friends. Real-time communication is another key component of social apps. After making a match, individuals have the option of initiating contact with one another. Being able to communicate with minimal latency and traffic across the web is essential.

The second is **gaming** apps. Matchmaking is a core component of forming both teammates and opponents. Poor matchmaking will lead to player frustration, especially if one is repeatedly outclassed by opponents, or one's teammates are consistently less skilled than the player. Moreover, real-time communication is involved in the ability of a game to support communication between players. When a player issues a command inside a game that results in an updated state, other players need to be able to view this updated state as soon as possible to ensure a seamless experience.

The purpose of this paper is to give not only an overview of matchmaking and real-time web communication, but also to propose and evaluate original matchmaking algorithms, and also discuss and evaluate implementations of real-time communication using existing web protocols.

II. PRIOR WORK AND RELEVANT TECHNOLOGIES

A. Matchmaking: No Stable Matching Exists

Consider a pool of users. What does matchmaking entail? It could be the task of finding, for each user, the best compatriot or opponent in the pool to match with, given the user's set of preferences. The best match for a single individual would be the one that he or she most prefers. However, the definition for the best match across a *group* of individuals is less clear. What happens if two individuals prefer the same candidate the most?

The **stable marriage** problem is a well-known "matching" problem that explores this challenge. The assumption in this problem is that that there are two sets of candidates, and these two sets of candidates are disjoint. Each candidate has a preference ordering that only involves candidates in the "other set"; in other words, candidates not in the same set as he or she.

An important idea to touch on is that this problem is solved, and there is an optimal solution. That is, there is an algorithm that produces a **stable matching** between candidates. In order to understand what entails a stable matching, let us first define what an **unstable match** is. Let x and y be two users in a matchmaking pool. A pair (x, y) is an unstable match if x prefers another individual y' over y, and y prefers another individual x' over x. Then, a stable matching is defined as follows: **a matching is stable if there is no such pair (x, y) for which the match is unstable**.

Gale and Shapley (1962) proved that in the stable marriage problem, a matching always exists. However, there is an issue when applying this problem to real-world matching problems. The underpinning assumption that there are two disjoint sets does not necessarily hold in general. This recognition gave rise to a variation that attempts to find a stable matching in the general case, known as the **stable roommates** problem. In other words, the stable roommates problem does not make the assumption of two disjoint sets.

It can be shown that a stable matching is not guaranteed to exist in the roommate variation. The following is a simple example of this fact. Consider a pool of four candidates {A, B, C, D}. Each candidate has ranked all the other candidates such that there is an ordered list of three candidates. Consider the following example. Letters appearing earlier are preferred (that is, A prefers B over C).

- A: (B, C, D)
- B: (C, A, D)
- C: (A, B, D)
- D: (A, B, C)

Consider then, all the possible matches that can be made given the pool. The possibilities are:

- (A-B), (C-D). Alternate (B-C) makes it unstable. That is, B prefers C over A, and C prefers B over D.
- (A-C), (B-D). Alternate (A-B) makes it unstable.
- (A-D), (B-C). Alternate (A-C) makes it unstable.

Note that for every possible combination of pairs, it is shown to the right an alternate pairing that indicates the matching is not stable [1].

Thus, in general matchmaking algorithms are faced with this particularly tricky challenge: a stable matching may not exist. The paper will discuss solutions to this challenge in depth in Section III.

B. Real-time Web Communication: Supporting Protocols

Only in recent years has real-time communication across the web become a relevant topic. This was largely due to the vast improvements and technological advancements made by modern browser and web technology. In the past, most websites did not have need for any real-time support. Most content was static, and dynamic content was served over other channels (such as Flash).

Nowadays, websites are increasingly demanding support for dynamic content, and the recent deprecation of Flash combined with many improvements to the Javascript language specification has led Javascript to become the desired language for animating content.

The popular HTTP protocol is used for allowing communication between a client and server across the web. HTTP is unidirectional, in that only the client can initiate a connection with the server via HTTP request. The server then provides an HTTP response. However, the server cannot in turn initiate an HTTP request to the client.

Then came the release of the WebSocket standard in 2011, which aimed to provide "full-duplex communication" across a TCP connection (RFC 6455) furthered this goal. This new WebSocket technology would allow a continuous connection for communication between a client and a server, with both parties able to initiate a message to one another on demand.

With two standards to choose from, this paper explores the relative speed (latency) between these two standards, and also the tradeoff in complexity of implementation in Section IV.

III. Matchmaking Algorithms: Assumptions and Objectives

It was demonstrated in Section II that there does not necessarily exist a stable matching in any given pool of candidates for matchmaking. It follows then, that there is no such algorithm that can return a "correct answer" to the matchmaking problem.

One solution is to not use "stable matching" as a correctness criteria. Rather, we define a different matchmaking quality measure, and then design algorithms to maximize that measure of quality.

The natural question that arises then, is what shall be the quality measure for matchmaking? In order to answer this question, we begin with a simplification by making a strong assumption.

One **fundamental assumption** that will apply for the remainder of this paper is that each match is made in a vacuum. That is, when attempting to match a specific candidate, we do not consider how a previous candidate was matched or how future candidates will be matched as a result of our decision. There are two justifications for this assumption:

- There is no guaranteed solution for the stable matching problem. Thus, attempting to optimize for the entire pool of candidates is not guaranteed to produce the global maximum. Moreover, this is an important assumption to make because it makes it possible to evaluate the algorithms described later against a "correct answer". We evaluate each match individually, as opposed to the matching of the pool as a whole.
- The pool of matches is constantly in flux. Whenever a new player joins, all the other previous matches need to be re-evaluated when optimizing over the entire pool of candidates.

This leads to the addressing of three related questions. First, what constitutes a "good match"? The author of this paper claims that the best matches, in the context of social networking, are ones that reflect compatibility. That is, the users enjoy interacting with one another. The justification rests on the assumption that users who engage in social networking want to enjoy interacting with others, whether that is in a friendly, professional, or romantic context.

The best matches in the context of a competitive game are ones that reflect equal skill. That is, the users are as close in skill level as possible. The justification rests on the assumption that players have more fun when they are equally matched in skill. A player who is greatly outclassed by his or her opponent will not enjoy playing the game. Similarly, the opposite situation – that the player is much more skilled than the opponent – is assumed to produce less enjoyment.

Second, we consider how many metrics to use when forming a matchmaking quality measure. That is, will the quality measure depend on a single factor or a number of factors? This is effectively determining the **dimensionality** of the inputs to the quality measure.

A. *Single-variable case*

This is the simpler case in which the quality measure depends on only a single factor. For example, this factor might be a measure of skill for each player. Then the solution is to simply minimize the difference in win-loss ratios to find the best match.

B. *Multi-variable case*

When considering more than a single factor or **feature** (a feature being a metric affecting the matchmaking quality), devising an algorithm for finding the best match becomes significantly more complex. This section will proceed to discuss two ways of handling such a situation: 1. Using a heuristic algorithm to minimize across multiple features, and 2. Implementing a quality measure function that maps a vector input into a scalar output of quality.

Third, arises the question of what are valid features to the matchmaking algorithm? This depends on the application of the algorithms, and tailoring of inputs to use cases and audiences. Indeed, the evaluation of matchmaking quality from tailored inputs vs. matchmaking quality of default inputs is encouraged. However, for the purposes of this paper, the author will assume that the factors affecting matchmaking quality are given (in a real-world context, this might be an analytics department). The algorithms presented are tasked with learning the relative importance of these weightings between features.

With these goals, assumptions, and considerations in mind, the author presents the proposed algorithms in the following section.

IV. Matchmaking Algorithms: varations

This section characterizes matchmaking algorithms in the specific context of games, though it can be easily generalized to other fields. First, we should consider metrics for comparing two players. One obvious metric is skill levels – how can we evaluate skill between two players? A common metric is **ELO**, which is most famously known for being used in chess. **ELO** is a weighted win loss ratio, where the player gains or loses ELO rating based on the difference between the player's current ELO and the opponent's current ELO.

The details of the rating system are largely left open to implementation depending on the use case. ELO is only used as one example of a potential rating system to measure the "skill level" between two players. Some games may find other metrics more useful for evaluating skill. As long as every player is evaluated by the same metric, the rating system has fulfilled its purpose – its main use is as a metric is to differentiate between players. What follows is a simple algorithm that evaluates along this single metric (we consider later more complex modeling using multiple metrics, or "features" as we will call them).

A. *Algorithm 1a: Exhaustive Search with respect to a Single Variable*

The algorithm is a simple procedure that assumes a dimensionality of the input is one (that is, the input for the quality measure is a single variable), and finds the best match based on it.

Consider the following description of the algorithm:

For each candidate, find in the remaining pool of candidates the opposing candidate with the lowest difference in ELO. Match these two candidates. Remove both candidates from the pool.

Time: $O(n^2)$

Note that this algorithm comes with it a set of problems. First, because it runs in $O(n^2)$ time, the problem is not particularly efficient. Popular modern applications can often have over a million players "online" at any given time, and as a result, given a pool of a million players, this algorithm would run for $(10^6)*(10^6) = 10^{12}$ iterations!

Second, the matchmaking pool is changing with time. Thus, during an exhaustive search, a new player might join the existing pool. Thus, the best match at the start of the iteration might not be the best match at the end of a single iteration.

B. *Algorithm 1b: Heuristic Search with Threshold with respect to a Single Variable*

The two problems of Algorithm 1a (exhaustive search) can be easily accounted with a heuristic modification. The issue that the algorithm does not scale can be fixed by having an ELO difference threshold. Instead of searching for the best ELO difference, we simply take the first one that meets a threshold. If the spread between ELOs inside the current pool is large, we can apply a scheme where we increase the threshold every d players searched, where d is a constant. We can further specify that after a certain number of players searched (for example 100). Then, our runtime is reduced to $O(cn) = O(n)$, where c is a constant.

This heuristic also resolves the second issue with exhaustive search: the matchmaking pool is constantly changing. Specifically, because the pool is changing, and there is no guarantee of a stable matching, it follows that a heuristic approach with faster runtime would not result in significant losses.

C. *Algorithm 2a: Heuristic Search with respect to a Vector*

This is a simple, naïve algorithm for finding the best match with respect to multiple variables. It begins by ranking the features from most important to least important. This can be done either randomly (if there is no information on relative importance of features) or based on prior knowledge.

Then, if there are k features, this algorithm will first find the k best candidates along the most important features. It proceeds to use each of the remaining features to prune. For example, it examines all k candidates by the 2nd feature.

Logically, this algorithm removes the worst of a certain feature in the entire candidate pool first. Among these worst for a certain feature, there may be candidates that are the best for another feature. Thus, by ranking the relative importance of the features, we are trying to find as optimal an order as possible for determining the opponent.

Some features we may want to maximize, other features we may want to minimize. Overall, our algorithm becomes a series of maximizations and minimizations along each feature. There is no guarantee of correctness from this algorithm, but the premise is that it works well for the right examples (we will demonstrate later in comparisons with other algorithms where this algorithm does not work well).

This algorithm takes $O(n + k) = O(n)$ if $k << n$ (the number of features much smaller than pool of candidates, which is generally the case).

What now follows is a discussion of the interpretation of finding a "best quality measure", given a feature vector of inputs. Concretely, how can we translate a vector of features into a scalar-valued quality measure?

The realm of machine learning provides potential aid in this area. Essentially, the goal is to learn a function that maps a vector-valued input into a scalar output, where the scalar output is the measure of quality.

Recall earlier the two assumptions that the best match in social networking is when there is compatibility, and the best match in games is when there is equal skill between opposing players. It follows directly, then, that the measure of match quality can be framed in the lens of maximizing the probability of compatibility, or maximizing the probability of equal skill.

D. Algorithm 2b: Logistic Regression for Binary Classification

The logistic regression algorithm in the binary case learns a function f that takes as input a vector of dimensionality n, and maps it to a label 1 or 0. It does this by calculating the probability of 1 (or 0), and then makes a decision based on which probability is more likely (the probabilities sum to 1). The mapping process works as follows: after determining a **weighting** for each feature (forming a **weight vector**), the algorithm proceeds to take the **dot product** of the **weight vector and feature vector**. To ensure that the output falls between 0 and 1, the dot product is then passed through a **sigmoid** function.

As Figure 1 illustrates, the larger the value of the dot product (the "more positive" it is), the more certain the algorithm becomes that this is a *good* match. On the contrary, the lower the value of the dot product (the "more negative" it is), the more certain the algorithm becomes that this is a *bad* match. Dot product values close to 0 indicate general uncertainty (where exactly indicates exactly 0.5 probability either way).

Why is this important for matchmaking? In other applications our focus may not be on the probabilities, but rather the predicted value. Why should we distinguish between 0.6 probability of a good match vs. 0.8 probability of a good match? These probabilities are useful for matchmaking because it allows us to evaluate candidate opponents relative to one another. Even if the regression algorithm predicts a good match for both candidates, one may be predicted with more confidence than the other. Naturally, the matchmaking algorithm picks the pairing of higher confidence.

Procedure: Train the logistic regression function on data from the user's history of "compatible" peers, and learn a function that calculates the probability of compatibility, given a *new, unseen* peer.

The function learned takes as input the feature vector of this new, "potential match", and outputs the probability of compatibility. Thus, given a pool of candidates, we can run each candidate through this learned function, and pick the candidate that **maximizes the probability of compatibility**.

Note that logistic regression requires a "training set" of data. A training set can be obtained from recording historical matches between players: for each match, store in a database the differences along the measured metrics between players, and also whether it was a successful match.

For example, Player A and Player engage in a game, where their ELO difference is 100 and their latency difference is 50. The match turns out to be a satisfactory pairing, and it is now stored in the database with these properties (100, 50, 1), where the final entry of "1" indicates this was a satisfactory pairing.

Figure 2 illustrates the process of training a logistic regression model based on a training set of data. In this case, logistic regression models the successful pairings as blue dots, and unsuccessful pairings as red dots. The algorithm essentially learns a "decision boundary" on which to classify a pairing as successful or not. Furthermore, the algorithm classifies points with different confidence. Clearly, when the algorithm is told to classify a point close to the line it is less certain in its prediction (center of picture), then when it is told to classify a point far away from the line (bottom-left or upper-right).

Logistic regression communicates this certainty, or

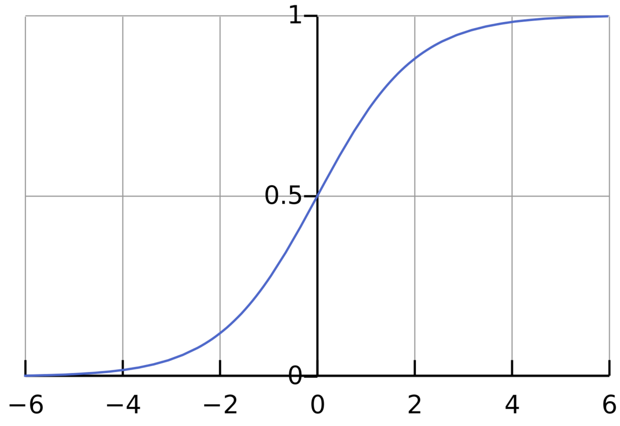


Figure 1: Sigmoid function.
(Source: Wikimedia Commons)

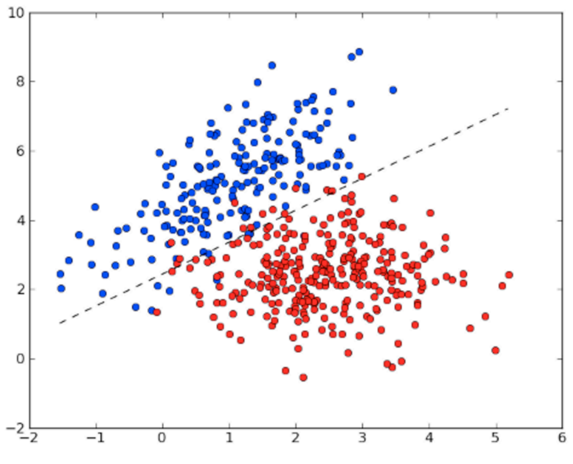


Figure 2: Training logistic regression
(Source: Stack Exchange)

"confidence" through a probability value. Specifically, it estimates the probability (0 <= p <= 1) that a point is classified as a 1. This is very useful in context of matchmaking, because we can use the probability value to distinguish between pairings that are "somewhat likely" to produce a good match and pairings that are "very likely" to produce a good match. For a more thorough treatment of logistic regression, please refer to [2].

In Section V we compare the quality of matches made via Logistic regression against the quality of matches made via Multi-variable Heuristic Search.

V. Results of Matchmaking Experiments and Discussion

A. *Acquiring a data set*

One key challenge was the lack of available, real-world matchmaking data. The most popular applications that conduct matchmaking often use proprietary algorithms, and their gathered data is proprietary.

Given this challenge, the author of this paper generated a dataset for the purposes of running experiments using the following procedure:

- Specify a **true weighting** of various features with regards to the match quality.
- Generate a dataset using these true weightings, and then purposely mislabel a certain percentage of the dataset to imitate **noise** in the real world.
- Once the data is purposely mislabeled, use this generated dataset to train the prediction-based algorithms. Note that the heuristic and exhaustive algorithms do not require historical data.
- Create a test set, and assign the "correct" match based on the true weightings of the features. Essentially the true weightings can provide the best match for any example. Then run each algorithm on it, and measure the error rate.

B. *Evaluating Algorithms*

The subsequent procedure is then performed to evaluate each algorithm relative to one another:

- *Assume*: Recall the fundamental assumption made earlier that matches are evaluated based on each individual match, as opposed to the entire pool of matches. Then, all algorithms pick a match independently of future matches. In other words, Person B is matched to Person A without consideration for the future matchmaking pool.
- Run each algorithm, and compare each algorithm's determination of the best match with the "true" best match.
- The observed dataset was generated via hidden weightings, thus there is a "correct answer" to evaluate against.

In this examples there are two features: **ELO difference** and **latency difference**. As before, the implementation details of these metrics are left largely open. Specifically, they depend on the application of the matchmaking, and are tailored to the use-case. A social networking app, for example, might use objective measures such as "number of mutual friends" and "percentage of similar interests" as features instead.

For the sake of completeness, however, here is an interpretation of the intuition behind these two metrics in the current example. Given ELO measures skill, a lower ELO difference indicates that two players are closer in skill level. Similarly, the latency difference conveys the relative delay between the two players: players who are able to communicate more quickly with the server have an obvious advantage. Thus, we want to minimize across both features.

C. *Results (Comparison with Heuristic)*

ELO Difference	Latency Difference	Probability of Good Match
100	-73	0.83
40	**-26**	**0.84**
170	147	0.69
-190	7	0.90
270	-47	0.73
130	70	0.75
-50	152	0.81
440	-54	0.63
110	-131	0.85
10	-65	0.86

Figure 3: Matches (logistic regression vs. heuristic)

Figure 3 demonstrates the results obtained from running logistic regression (yellow) and running the multivariable heuristic algorithm (bold). Note that the "correct answer" is also in bold. The heuristic algorithm selected the correct answer of (40, -26).

Here is a high-level rundown of how the algorithms arrived at their results:

- Logistic regression, based on the weightings it learned from the training set, evaluated the "correct match" with 0.84 probability, and thus picked another match with higher probability (0.90) over it.
- Multivariable heuristic search first finds the best k examples along the first feature. In this case k = 2, given there are two features, and ELO difference is the more important feature (assume this is given). Thus it finds the two candidates with the lowest ELO difference. Then, (40, -26) and (10, -65) remain. It optimizes again, this time minimizing across the second feature: latency difference. -26 is less than -65, thus it picks (40, -26) as the best match.

Clearly, the heuristic algorithm matched correctly with the right candidate, whereas the logistic regression did not. In this particular use case, multivariable heuristic search is the winner. However all is not loss for the more complex logistic regression model. It is easy to demonstrate cases where heuristic search might fail.

- Again using ELO difference and latency difference, consider a new set of candidates: (1, 10000), (30, 30), (10, 1000).
- Suppose the weightings are approximately 70% importance to ELO and 30% to latency. Logistic regression obviously picks (30, 30).
- Multivariable heuristic search optimizes across first k = 2 examples. It picks (1, 10000) and (10, 1000). It then picks the lower latency difference in the two remaining examples, which is (10, 1000). Considering we want to minimize across both features, clearly the heuristic algorithm was the wrong choice.

Admittedly, the above example is contrived. Nonetheless, it is indicative of a flawed design with the multivariable heuristic algorithm. **The problem is that the heuristic algorithm optimizes each feature independently of all the others, whereas logistic regression optimizes across all features at the same time.** Logistic regression also has the added benefit of learning the exact weightings for each parameter. For example, how important is the ELO difference relative to latency difference? The heuristic algorithm ranks them by importance, but only encodes that one is better than the other (without describing how much better).

D. Results (Confidence of Correct Answer)

The following table presents the results of running logistic regression matchmaking run on a dataset of ten different users during a **live** running of the app. It presents both the correct answer, based on underlying true weightings during generation of the data, and also the weighted guess.

Table 1: Real-time "live" matches made via logistic regression, and measure of correctness ("gap")

Logistic match	"Correct" match	Gap
(770, 101) ⬄ (970, 203)	(770, 101) ⬄ (810, 119)	0.165
(620, 270) ⬄ (920, 200)	(620, 270) ⬄ (550, 257)	0.234
(850, 230) ⬄ (850, 218)	(850, 230) ⬄ (850, 218)	0.000
(920, 200) ⬄ (810, 119)	(920, 200) ⬄ (970, 203)	0.063
(540, 249) ⬄ (710, 106)	(540, 249) ⬄ (710, 106)	0.000

⬄ denotes match

The "Gap" column is the probability gap between the logistic match and correct match. Essentially, the logistic regression algorithm evaluated the probability of both potential pairs, and the gap is how much "better" the logistic algorithm found the actual match (the match it chose to make) than the correct match (according to the true weightings). It is a useful indicator of how much logistic regression leaned towards making a certain match over the correct match.

Specifically in the first example logistic regression had a probability gap of 0.165. This means that logistic regression was approximately 16.5% more certain of the "logistic match" than the "correct match".

What leads to these errors? One important fact to consider is that one wrong mismatch can lead to propagating mismatches. Because two candidates are taken out of the pool, future matches are certainly affected.

Recall the earlier assumption that matches are made in a vacuum, given there is no guarantee of a stable matching. If our algorithm was given the true weightings, we could match perfectly. However, because logistic regression **guesses** the true weightings, and makes matches based on the best guess. The algorithm is imperfect.

How can we interpret these errors meaningfully? We can observe that the confidence gaps are significantly less than that of random guessing. First, note that 2 out of 5 matches were made correctly (the gap is 0). Second, the matches that were wrong had relatively low gaps. The highest was 0.234. In a random matching scenario, we would expect gaps consistently around 0.25 (given that 0.5 is the probability score of complete uncertainty).

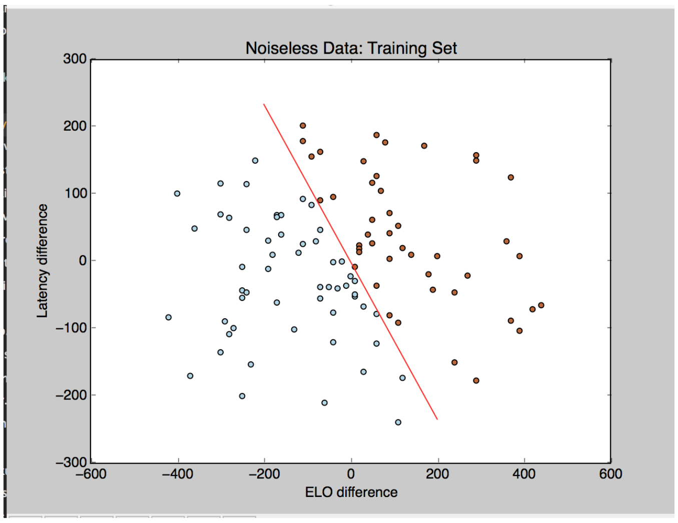


Figure 4: Training on noiseless dataset

E. Tradeoffs Between Heuristic and Learning Approach

What then are the tradeoffs of using the logistic regression model? The first is the **historical data prerequisite**. Logistic regression requires a training set with a reasonably large number of examples to train and construct a model. If such data is not available, then this is not an option.

The second is **overconfident predictions**. An interesting anomaly that arose during the evaluation of algorithms was astoundingly high confidence of predictions by the logistic regression algorithm.

Consider the training set Figure 4, generated by the author. There is a clear line separating the positive and negative training examples (the dots that predict a good match and the dots that predict a bad match). Thus, the trained model, represented as the **decision boundary** (the formal term for the separating line), can predict every single dot in the dataset correctly.

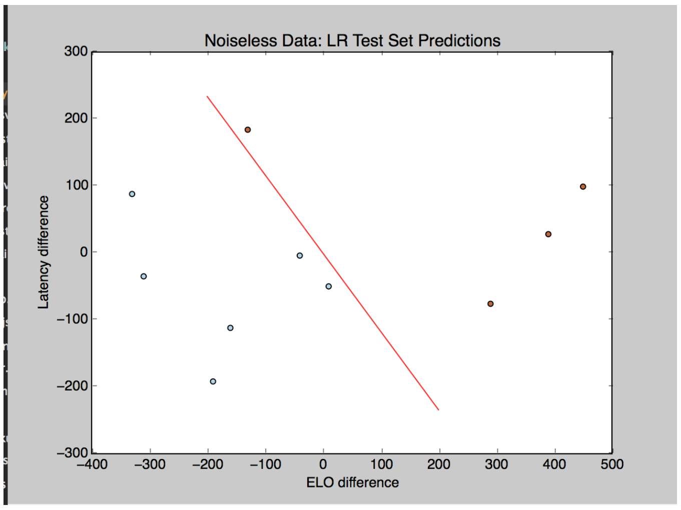


Figure 5: Testset predictions

This leads to overconfident predictions on the test set. Specifically, almost all predictions were made with probability extremely close to 1 for good matches, and probability close to 0 for bad matches. Figure 5 graphs the predicted values. Observe again the boundary line that perfectly separates the two classes of examples.

The source of this anomaly lies in how the weightings are calculated (or "trained"). **Specifically, the weight vector norm tends towards infinity** in the separable case. In other words, the weightings for each feature will be set to enormously high values. This problem occurs because of the method through which the logistic regression algorithm is trained. The weight vectors are optimized via an iterative gradient ascent algorithm. The objective is to maximize the likelihood of the "parameters" (that is, the weights), given the observed data. In the separable case, no examples are misclassified, so there are no penalties for higher weight values. As a result, the algorithm will arbitrarily scale the weight values to increase the likelihood, and they will tend towards infinity.

A key insight from this anomaly was that the initial generated dataset contained a problem. Specifically, the initial dataset generated by the author did not contain noise (misclassified examples). Thus, it is relatively easy to see why the data was linearly separable, and resulted in this problem. In real-world datasets containing random noise, a linear separation line that classifies every example correctly is less likely. As a brief digression, when there is a dataset that exhibits linear separability, it can be resolved by introducing regularization [3].

Realizing the flaw, the author of this paper introduced random noise to the dataset, which arguably increased the likeness of the synthetic data to real data. The results presented earlier Section C were from training on synthetic data with noise.

Figure 6 graphically depicts the generated dataset, with added noise (purposely misclassified examples). Note the lack of a separating line – specifically no linear decision boundary can be drawn that perfectly separates all examples.

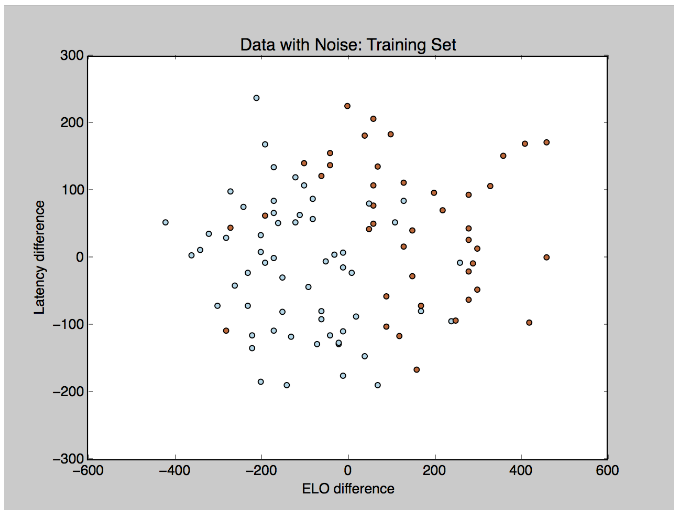


Figure 6: Dataset with noise

VI. Alternative Learning Algorithms

Logistic regression certainly is not the only algorithm for learning weightings given a set of inputs. Logistic regression generally assumes a real-valued feature vector as input (though it is possible to supply discrete-valued inputs). However, there are algorithms better suited for the discrete case. Specifically, Naïve Bayes is an algorithm that takes in a vector of discrete-valued inputs (every value in the vector is drawn from a finite set of classes (e.g. {0, 1}), which might apply better in certain use cases. For example, consider a matchmaking algorithm for determining compatibility between love interests. Pairing two people might be based on a set of answers to yes or no questions, e.g. "Are you a smoker? Do you enjoy swimming?" The answers to these questions can be encoded as a discrete-valued (binary, in this case) feature vector, rather than real-valued.

Naïve Bayes, similar to logistic regression, outputs the probability that there will be compatibility given these binary vectors. The means through which it calculates this probability is quite different however.

Moreover, there are other algorithms that take as input a feature vector of real-values and output a classification label (0 or 1 in the binary case). These algorithms might involve techniques to force the separability of two classes (e.g. the blue and red points) by mapping to a higher dimensional space, or by drawing a nonlinear decision boundary (a curved line separating the two colors of dots). The literature on learning algorithms is extensive, and the field is constantly expanding.

VII. Real-time Web Communication: Minimizing Latency And Traffic

This paper will now shift and discuss the other topic: real-time web communication. The main goal of this section is to discuss how to minimize latency between clients that wish to speak with one another across the web. Specifically, it will discuss two ways to implement real-time web communication, and their tradeoffs.

This paper does not explore data integrity across a communication channel. In fact, the implementations presented by this paper are based on existing protocols, in which data reliability is abstracted.

A. *Polling with HTTP*

The key limitation of HTTP is that the client must be the one to initiate the request-response cycle with the server. The server cannot send a request to the client. Thus, for one client to receive updates about another client's action, it needs to first make a request for that data from the server.

The first implementation of real-time between two clients uses polling. Specifically, it uses the HTTP protocol to simulate communication between two clients by polling a server. The following description of the implementation assumes that two clients are playing a game together, but it can be easily generalized to other applications that use real-time (such as chat). The scheme works as follows:

- Client A and Client B are matched into a game.
- Client A makes a move by sending an HTTP request to the Server.
- The Server receives the request, and generates an HTTP response. If Client A's move was valid, the Server responds with an updated move to Client A.
- The Server expects an update request from Client B. However, it cannot tell Client B directly, because the interaction must begin client-to-server.
- Thus, Client B *polls* in a loop, so that every x seconds, it makes an updated request from the server. Client B can also ask for an update request before every move – that is, if it wants to make a move, it first makes sure it has the most updated state of the game before moving.

B. *Bidirectional Communication with WebSocket*

The WebSocket protocol, unlike HTTP, allows initiation of communication between both the Client and Server. Moreover, the connection is maintained as open, whereas in HTTP there is the overhead of starting a connection on each request.

The following implements the same game described earlier using WebSocket:

- Client A and Client B are matched into a game.
- Client A makes a move by sending a message via WebSocket to the Server.
- The Server receives the request, and then emits an *event* to all players in a game. In other words, it *initiates* a message to Client B to let it know that Client A made a move.
- Client B then can respond appropriately to the Server, on receiving this update.

Client B no longer needs to poll in a loop to get an update on A's latest move (nor vice versa). Instead, it receives updates directly from the server, and gets the new data as soon as it's available.

VIII. Results of Real-Time Communication Experiments and Discussion

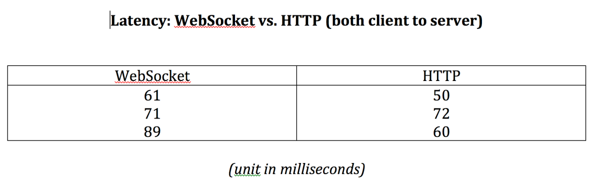
A. *Methodology for Comparing HTTP and WebSocket*

HTTP and WebSocket were compared by evaluating the length of time it would take for a message to reach the server from the client. In HTTP, this was done by making a POST request, in which the message was stored in the body of the request. In WebSocket, this was done by emitting a socket event that directly sent a message to the server via an open connection.

- In order to measure the elapsed time from the client's sending and server's receipt, the only data stored inside the body would be the current time. In Javascript, this can accessed through Date.now(), which gets the current epoch time in milliseconds.
- Then the server, upon received the message, would call Date.now() and subtract from it the time value stored in the body of the message. The resulting value is the difference in milliseconds between the send and receipt of the message. Measuring across three trials, the values are then presented as the measured latency.

B. *Results from Comparison*

It seems relatively intuitive that the WebSocket data would present better results for a real-time use case. Yet the results from Figure 7 above demonstrate that this was not the case. As the table below clearly demonstrates, the latency of both WebSocket and HTTP were approximately the same.



Latency: WebSocket vs. HTTP (both client to server)

WebSocket	HTTP
61	50
71	72
89	60

(unit in milliseconds)

Figure 7: Latency measures in HTTP vs Websocket

C. *Tradeoff: WebSocket vs HTTP*

Though Figure 7 illustrates that the results between WebSocket and HTTP are similar, WebSocket is still superior a number of ways. First, there is benefit in allowing the server to initiate connections with the client. To quantify this, the number of messages can be used as a measure of load. For example, a game state update using the HTTP scheme requires two messages: one from client to server, and one from server to client. On the other hand, a game state update using the WebSocket scheme only requires one message: server to client.

Based on the first trial, a one-way connection takes 50 milliseconds in the HTTP case, and 61 ms in the WebSocket case. However, because a single update on the status of other clients might require multiple messages in the HTTP case, but only a single round of communicate in the WebSocket case, WebSocket has the advantage with respect to total latency.

Figure 8 emphasizes this with a diagram that shows where the advantage of WebSocket over HTTP lies [4].

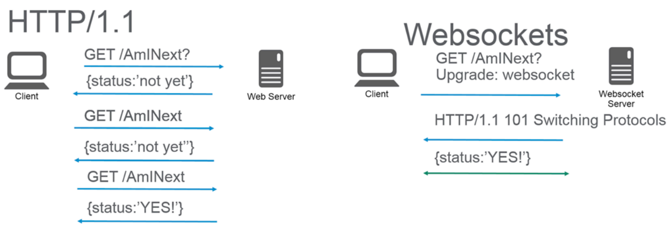


Figure 8: HTTP vs Websocket Diagram

IX. Forming a Framework

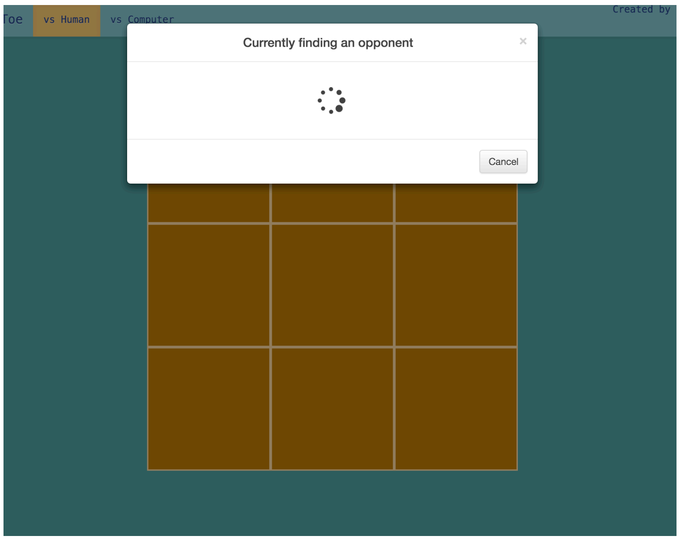
A. *Framework and Sample Apps*

Specific implementations of the matchmaking algorithms, and the real-time web application are provided on GitHub.

A developer using the framework has the option of tailoring matchmaking algorithms to their own preference. For example, if developers do not have historical data to train a logistic regression model, they should configure the framework to use the heuristic algorithm.

Finally there are samples provided by the author, and furthermore, an empty "framework", that developers can clone and proceed to fill in with their code that has the core of matchmaking algorithms built in (which the developer can tailor for the specific app they envision).

Figure 9 and 10 are two illustrations of a sample tic-tac-toe app in action.



Currently finding an opponent

Figure 9: Matchmaking in progress in a sample app built with the framework

X. Supporting Technologies and Implementation in Practice

This paper has so far discussed both matchmaking algorithms and real-time communication, and demonstrated the performance and tradeoffs of various implementations of each. These pieces effectively form a framework for building a modern web application with accurate and efficient matchmaking and real-time web communication. As a final point of discussion, it is worth mentioning the technological toolkits that enable this framework to work.

A. *Implementing Matchmaking in Practice*

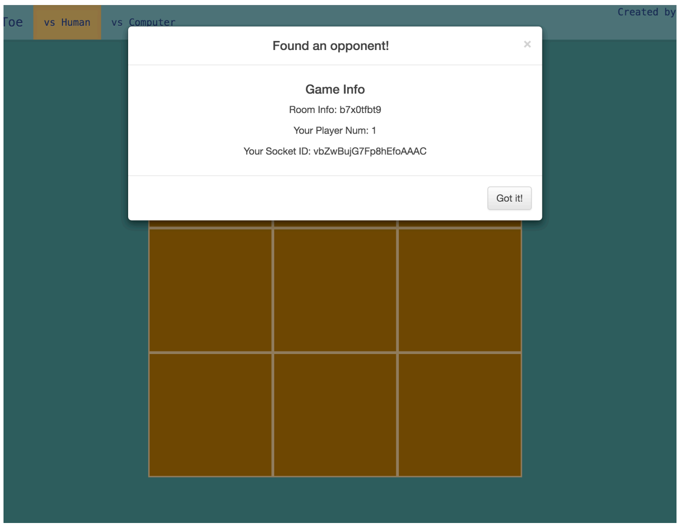
Algorithms 1 and 2 (exhaustive and heuristic algorithms) are relatively easy to implement. Indeed, it amounts to finding the minimum, or checking if a value falls below a certain threshold.

The learning algorithms can be implemented using a standard machine learning library. Specifically, the author used scikit-learn (Python machine learning library) to implement a logistic regression based classifier that also outputs the confidence (probability) of predictions.

B. *Implementing Real-time in Practice*

HTTP web applications are quite popular, and there are a large number of web frameworks written across many programming languages for implementing a web server. Particularly popular is Node.js, which is a framework that processes client requests asynchronously [5].

In addition, the growing popularity of real-time applications has led to utilities that specifically implement WebSocket. Specifically, there is socket.io and Django Channels as two utilities for implementing WebSocket, written in Javascript and Python respectively [6].



Found an opponent!

Game Info

Room Info: b7x0tfbt9

Your Player Num: 1

Your Socket ID: vbZwBujG7Fp8hEfoAAAC

Figure 10: After a successful match in a client app

C. *Client-side Technology Conducive to Real-Time*

Web applications with real-time communication support most likely demand rapid updates to webpages, and fast processing of animations. The standard model of editing the DOM is slow and expensive for browser processing.

Modern client-side frameworks can alleviate the issue. In particular, a new rendering engine known as React, provided by Facebook, is specifically designed to allow for fast updates to webpages.

XI. Acknowledgements

I would like to thank Sean Smith and Lorie Loeb for being a part of my thesis committee and offering feedback on my thesis.

Moreover, I would like to thank my advisor Xia Zhou for providing guidance throughout these past two terms. This has been a tough and challenging, but fulfilling project.

References

[1] Irving, Robert. "An Efficient Algorithm for the Stable Roommates Problem." *Journal of Algorithms,* 6: 577-595.

[2] Bishop, Christopher. *Pattern Recognition and Machine Learning*. New York: Springer, 2006. Section 4.3.2.

[3] Bishop, *Pattern Recognition and Machine Learning*. Section 3.1.4.

[4] Chen, Eric. "Load Balancing WebSockets." Technical Article: F5 Networks.

[5] Node.js Foundation. https://nodejs.org/en/.

[6] Socket.IO. http://socket.io/.